Integrating protocol aspects with software components to address dependability concerns

Andrés Farías
NCX
1945, Av. Providencia, Office 303
Santiago, Chile
andres.farias@gmail.com

and

Mario Südholt
INRIA/École des Mines de Nantes
OBASCO group
École des Mines de Nantes
4 rue Alfred Kastler
44307 Nantes Cedex 3, France
sudholt@emn.fr

Abstract

Dependency properties of software systems, such as availability and safety, typically cannot be modularized using traditional means for encapsulation, such as objects or components. Programmatic means for such concerns developed in the domain of Aspect-Oriented Programming (AOP) should thus be applicable. However, current systems for AOP do not meet important needs of dependability concerns, especially because of insufficiently expressive aspect languages and lack of reasoning techniques about aspect-oriented programs.

In this paper we address these problems by introducing a notion of aspects for an existing model of software components which include finite-state protocols in interfaces. We present two main contributions. First, an aspect language enabling modification of the structure and the dynamic state of protocols. Second, a discussion of the preservation of basic compositional properties of components modified by aspects and how recent static analysis techniques for aspect interactions carry over. Finally, we illustrate the usefulness of our model in the context of dependability properties of a real-world application.

Keywords: aspects, software components, finite-state protocols, dependability properties, aspect interaction analysis

Technical report no. 04/6/INFO

École des Mines de Nantes

November 2004

Contents

1 Introduction

Research on dependable systems addresses the problem of making explicit dependencies between systems (including software systems), principally properties such as availability, reliability, safety and maintainability. On the programming level, such dependency properties typically correspond to concerns who cannot be reasonably modularized using traditional means for encapsulation, such as objects or software components. Programmatic means for the handling of such concerns the principal subject matter of the domain of Aspect-Oriented Programming (AOP) [15]. An application of aspect-oriented techniques in the context of dependable systems is therefore potentially valuable and first steps in this direction have been taken (see, *e.g.*, [13]).

However, most aspect-oriented languages and tools are subject to two shortcomings which are especially problematic in the context of dependable systems. First, many dependability issues are most naturally described in terms of sequences of execution events (*e.g.*, vulnerabilities yielding to faults causing subsequent errors, cf. [5]). However, the so-called pointcuts of current AO languages, which define the execution points to which aspects can be applied, are most frequently restricted to (sets of) individual execution points, making the definition of dependability concerns in terms of sequences at least awkward. Second, almost all current AOP systems are geared towards the expression of aspects but do not provide any means to verify properties of AO programs. This is particularly problematic *w.r.t.* fault prevention, tolerance, and removal properties.

In this article, we present an integration of a notion of aspects which addresses these two shortcomings in the context of software components which include finite-state protocols in their interfaces. Concretely, we present three contributions. First, we present an aspect language for the manipulation of protocols of such software components and formally define the aspect language by instantiating a recent generic framework for regular aspects [6]. Second, we discuss three basic composition properties and show how several static analyses for interactions among aspects can be adapted to aspects over components. Finally, we illustrate how this model of components with explicit aspects can be usefully applied to tackle dependability properties of a real-world application.

The paper is structured as follows. Section 2 presents the component model with regular protocols our work is based on and the application whose dependability properties we want to improve. Section 3 defines the syntax and semantics of our aspect language. In Sec. 4 we discuss basic composition properties and analysis of aspect interaction properties. Section 5 presents related work. A conclusion and further work is given in Section 6.

2 A component model with explicit protocols

In order to investigate an explicit notion of aspects for component-based programming, we first briefly present the CwEP component model [11, 10] on which we base our investigation.

Since aspects are intended to modify a base application's behavior, AO languages essentially rely on means to refer to the base applications code or execution. Three characteristics of software components are of particular interest for the integration of aspects in this context: enable aspects to be defined in terms of complex component behavior, provide explicit support for typical communication patterns, and provide an abstraction for component identities (e.g., to support actions related to fault tolerance and removal as well as for security purposes).

The CwEP model supports these characteristics through the incorporation of finite-state protocols into component interfaces. Protocols allow much richer component behavior to be exported than allowed by interfaces consisting simply of method signatures (which is the case in almost all current component models). Protocols also allow to capture many typical interaction patterns among components. Furthermore, we allow protocol transitions to be constrained by sets of component identities, e.g., in order to allow some transitions to be taken only by explicitly-defined components, *e.g.*, to react on an error after a certain sequence of requests from a given set of other components have been performed.

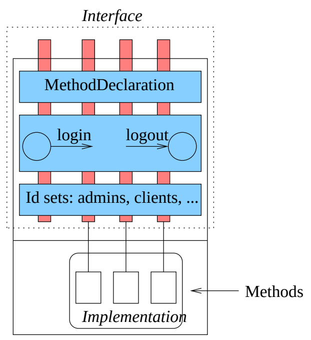

Figure 1: Structure of components with explicit protocols

Concretely, Figure 1 shows the basic structure of components with explicit protocols. A CwEP component defines an interface composed of three parts: a set of method declarations, an interaction protocol and a set of identities. First, the set of method declarations defines the set of methods offered and implemented by the component and corresponds to the notion of interface common in object-oriented and component-based settings. Second, the interaction protocol is a finite-state automaton defining transitions between a component and its collaborators. Third, identity sets may be used to label protocol transitions based on identities of collaborators (represented, *e.g.*, using public-key certificates). Specifically, transitions can be annotated using four operations/predicates involving identity sets:[1]

$x+$: Add the identity of the component performing the current transition to set x.

$x!$: Enable the current transition only for components whose identities are in x.

$x-$: Enable for components in x and remove the identity of the component taking the current transition from x.

$x*$: Execute the current transition once for each identity in x.

In order to illustrate this concepts we refer a real-world application called *SIS 2004*. This application allows doctors of government hospitals to register information about patients and diseases. Figure 2 shows a simple protocol which defines part of the interaction between clients and the server of our example. From the initial state (state 0) a client can connect to the server, in which case s/he is added to the identity set u. From state 1 only users in u (note the label $u!$) can lock the case of a patient. In state 2 the server broadcasts to all clients (expressed through the label $u*$) the fact that a case of a patient has been locked. Finally, from state 2, the client can disconnect from the server and its identity is removed from the clients identity set u (label $u-$).

2.1 Composition properties

A major problem of component-based programming is the analysis and/or enforcement of properties of composed software. Two properties are of fundamental importance in the con-

[1]A proof that finite-state automata whose transitions are thus labeled remain finite-state (under the hypothesis of a finite number of identities) is given in [10].

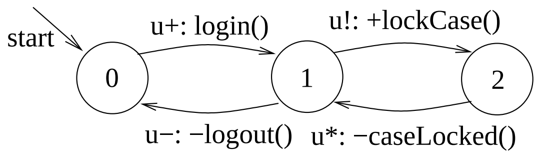

Figure 2: A simple protocol for client-server based communication

text of component-based software: substitutability and compatibility.[2] If a component c_1 is substitutable for a component c_2, c_1 can be used in all contexts in which c_2 is applicable. A component is compatible to another one, if communication requests by the former can always be satisfied by the latter.

These properties become much more intricate for components whose interfaces include protocols compared to components whose interfaces consist of sets of method signatures only. We use one of several proposals for the definition of substitutability and compatibility of protocol-based components which is defined in terms of traces, which define the requests which can be handled throughout execution of a protocol, and failures, which specify requests which cannot be handled in a specific state of the protocol. (See [11] for the definition used in CwEP and pointers to related work on this subject).

2.2 Component composition through protocol composition

The CwEP model features protocol composition operators to facilitate component construction and assembly. In [11] operators have been formally defined for the concatenation of protocols, adding new branches to protocols and insertion of a protocol in another one. For instance, new branches defined as a protocol p_1 can be added to a state s of a protocol p_2 using the "union" operator, which, informally speaking, merges s with the start state of p_1 so that any possible transition from p_1 or p_2 can be taken from s.

Preservation of compositional properties involving substitutability and compatibility have been proved for such operators. For instance, any protocol which can substitute one of two given protocols can also substitute the union of the two protocols (see [10] for a proof). This makes these composition operators particularly interesting for component-based software development: once basic properties have been proved (most frequently manually), properties of composed applications can be ensured by construction.

2.3 SIS 2004: tracking data about progress of diseases

We now describe in some detail the medical 3-tiers application *SIS 2004* to which we applied our method in order to modularize three dependability concerns.[3] This application was built on Sun's J2EE platform for the health department of the Chilean government and is used today in every public Chilean hospital. This application is used in hospitals in order to control the progress of certain diseases. In particular, the Chilean government ensures short delays for treatment at some critical steps of diseases. For example, for cataracts the patient is ensured that no more than 6 months pass before a certain treatment is applied. The overall

[2]Note that we do not consider here farther-reaching properties, such as deadlock properties.

[3]A more detailed presentation of the application is given in the companion technical report [12].

progress of the illness can be represented as an automaton that states the possible orders in which disease-related events can take place.

Figure 3 illustrates part of the protocol for the events registered for the cataract illness. This protocol embodies a complex interaction between the server and the client. The client submits some events for cataract patients and the server confirms requests and initiates some actions. The shown protocol states that a doctor must first register a suspicion of a patient having a cataract. The server confirms that a case has been created and that other events can be registered now. The patient may be referred to other establishments in order to receive some medical attention such that another doctor can help in the diagnosis. If the diagnosis confirms the illness the patient's case is considered as confirmed and the treatment is pursued. Otherwise, the case is closed. After the treatment takes place the case can be closed (whether the patient's health improves or not, though). At every event registered by the doctor, the server confirms whether the event has been properly added to the patient's case.

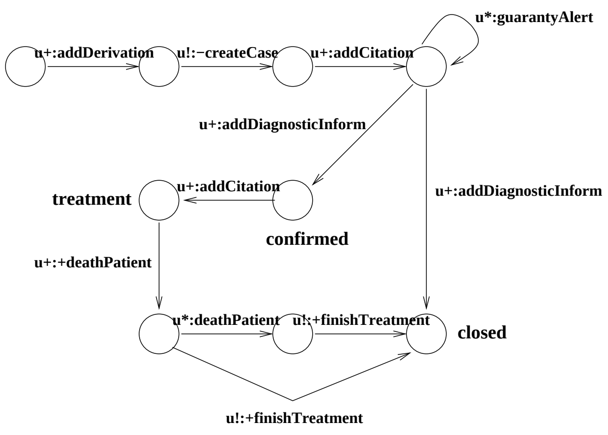

Figure 3: The protocol of the cataract illness.

A major problem in designing and implementing this application constituted different dependability-related functionalities, in particular failure handling in the context of lost sessions due to web server problems, maintainability problems caused by strong dependencies between the business and database layers, and security handling using a role-based security approach. We will show how to address those issues using our integration of aspects and components later on.

3 An aspect language for CwEP

In the following we present the the first contribution of this paper: an approach to modularize functionalities that crosscut components and apply it do the dependability concerns of our example application SIS. The CwEP model is appealing for this endeavor because it exposes part of the internal behavior of components through their interfaces in form of explicit protocols and manipulation of identity sets. Expressive aspects can therefore be defined using this information without access to the component implementation itself, *i.e.*, only using black-box type access to components.

Since CwEP components are based on finite-state protocols, it is useful to apply the same restriction to aspects defined on components. This raises, in particular, the possibility to adapt and apply reasoning techniques for finite-state automata to component systems which are subject to modification by aspects. In this section we present a language for such regular aspects to modularize crosscutting behavior over CwEP components. The language is based on a general framework for regular aspects [6, 7]. This framework provides a formal definition for regular aspects, defines two types of aspect weavers and features a set of static analyses

$$
\begin{array}{lcll}
P & ::= & A \parallel \ldots \parallel A & \\
 & & & \\
A & ::= & \mu a.A & \\
 & \mid & C \,\triangleright\, Ac \,;\, A & \\
 & \mid & C \,\triangleright\, Ac \,;\, a & \\
 & \mid & A \,\Box\, A & \\
 & & & \\
C & ::= & T \;\mid\; C.C \;\mid\; C_1 \talloblong C_2 & \\
 & \mid & C\{int\} \;\mid\; C_1 +\; C_2 & ;\, \mathit{first}(C_1) \neq \mathit{first}(C_2) \\
 & & & \\
T & ::= & (S,\ (Id, TI, M),\ S) \mid !T \;\mid T \ \mathit{and}\ T & \\
 & & & \\
Ac & ::= & Ac \,;\, Ac & \\
 & \mid & \mathit{unify}(P, S) & \\
 & \mid & (\mathit{add} \mid \mathit{remove})(Id, \mathit{IdSet}) & \\
 & \mid & \mathit{send}(M, Id) & \\
 & & & \\
P & ::= & \text{// finite-state protocol with identity sets} & \\
 & & & \\
S, Id, TI, M, \mathit{IdSet} & ::= & \text{// constants or single-assignment variables} &
\end{array}
$$

Figure 4: Syntax of regular aspects over CwEP components

for interaction properties among aspects. By instantiating and extending this framework, we provide a formal definition for aspects over CwEP components and provide static analyses for interaction properties of such aspects.

Aspect languages are commonly defined in terms of two sublanguages: a so-called pointcut language defining when (where) an aspect modifies the execution (code) of a base program and a so-called advice or action language defining what modification to apply if a pointcut is encountered. In our case the base program executions consist of execution of components constrained by finite-state protocols. our pointcut language permits to define execution points occurring after regular traces. Finally, our action language allows the protocol state of a component to be modified.

3.1 Syntax

Figure 4 defines the syntax of regular aspects over CwEP protocols. Note that the language is not intended to be used for programming, but is chosen to facilitate precise formal definition of its semantics and properties involving aspects and components. This language is based on the framework for regular aspects introduced in [6] and shares the main characteristics of that framework: an aspect program consists of a number of aspects which are applied in parallel to a base program. The non-terminal A derives regular aspects and has the same definition as in the original framework [6]: repetition (μ), sequencing of basic rules (;), and a choice operation of aspects ($\Box$, where the first aspect has priority if both crosscuts of the first basic rule of the aspects of the choice match). Aspects define finite-state expressions over basic rules $C \rhd Ac$ where C (a "crosscut") represents a pattern matching join points, such as service requests in a protocol, and Ac an action, *e.g.*, a call to a service of a component, which modifies the base program execution when the crosscut C is matched. Variables which are used in patterns are single-assignment in that they are bound by the corresponding value of a join point if they are still unbound, but have to match the corresponding value of the join point if they have already been bound.

We introduce two extensions of the original framework:

- A set of actions for the manipulation of finite-state protocols, both their structure as well as their dynamic state.
- A syntax for the definition of regular expression like crosscuts. Note that such regular expression crosscut define sequences of crosscuts which is different from regular aspects (already present in the original framework) which specify sequences of basic rules.

The first extension is the major feature of our aspect language necessary in order to manipulate component protocols. The second provides syntactic sugar facilitating aspect definitions.

Concretely these extensions manifest itself as follows. The non-terminal C extends the previously-defined framework by regular expressions at the crosscut level. The different alternatives are respectively basic terms, concatenation (.), a logical or ($\|$), and two repetition operations, one yielding a fixed number of repetitions ($\{i\}$), the second repeats one or more times followed by a concatenation ($C_1 + \ C_2$). Note that this set of regular expressions is restricted compared to the common definition (e.g., no Kleene star '$*$' operator) because we have to avoid non-deterministic expressions, in order to be able to give a reasonable definition of aspect weaving. The non-terminal T represents basic terms, which come in three variants: protocol transitions represented by triples (start state, transition labels, end state) as well as conjunctions and negations of basic terms. The non-terminal Ac defines actions, which are sequences built from three basic actions: an operation which modifies the protocol structure by unifying two protocols; an operation adding the dynamic protocol state by allowing addition/removal of identities; finally, an operation inserting a "one-time", i.e., temporary, new transition.

3.2 Aspects for the handling of dependability concerns in SIS 2004

We have mentioned three dependability concerns of the *SIS 2004* medical application in the previous section. In the following, we show how our aspect language can be used to address such issues.

1. *Error treatment for lost sessions.* Due to a problem existing in several web server's implementation, it can happen that client sessions can be lost, or worse than this, yielded to another client. This problems specially apear when clients are behind a firewall and the server session management is based on the client IP address. In this case, it is extremely important to ensure that a client has not get a wrong session because it could be used by an user to impersonate another user with higher privileges.

   Let us illustrate how to tackle (part of) this dependability problem using our aspect language. In order to prevent this erroneous situation, we can define a sequence of interactions between the client and the server which ensures that the client has correctly obtained a session. An appropriate aspect for this guard mechanism can be defined as

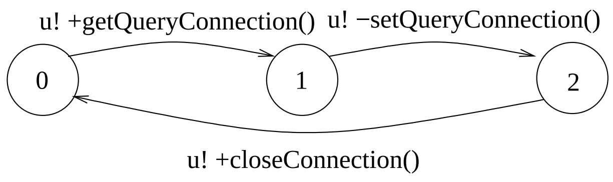

Figure 5: Protocol s for clean termination of a database connection.

follows (note that we use the name of a request as a shorthand for the corresponding transition if no ambiguity is involved):

$$\mathit{GuardSession} = \mu x.\texttt{getSession()} \rhd \texttt{setSession(getValidSession())}; x$$

This aspect is defined by using two different methods: `getSession()` is the standard method used to obtain the request session. This is however a not safe method for the use of it may lead the client with an invalid session. The problem was solved in the *SIS 2004* application by programming a safer method called `getValidSession()` which ensures that the client that use the method will always obtain a valid session. The above aspect ensures that every time a client uses the potentially failing method `getSession()` the server will reinitialize the current session by calling the method `getValidSession()`.

Furthermore we could ensure that this additional test is not performed if superfluous, *e.g.*, if `getValidSession` has already been called using the following expression:

$$\mathit{OptGuardSession} = (\mathit{getValidSession}; \mu x.y))\Box \mathit{GuardSession}$$

which prevents execution of session guards when `getValidSession` has previously been called. Technically, this is done by exiting the choice expression if `getValidSession` has been called by means of the immediately terminating aspect $\mu x.y$ in the first branch.

2. *Maintainability.*

   One of the most important constraint for the application is that response times must be very short. The *SIS 2004* was initially defined to have a good performance in both data transaction and data management. Therefore an important part of the business logic had to be defined in the database layer, introducing a strong dependency relation between the server and the database service. This service is used both to acquire entity objects (following the J2EE standard) and to perform other functionalities such as the construction of reports or to add medical events for diseases. This dependencies make the application much more expensive and difficult to maintain than an ideal 3-tiers application.

   A specific problem in this context is that frequently many connections to the service pool of the database are established (indeed, this number is proportional to the number of clients connected) which is expensive for the server. If the connections are not properly closed the application may stop responding to some functionalities. To manage this dependency, we define a protocol s, as shown in figure 5, that ensures that all connections to the database service are closed. We can ensure this by means of the following aspect:

$$SingleConnection = \mu y.(\_, \texttt{getQueryConnection()}, i^{Qu}) \triangleright unify(s, i^{Qu}); y$$

by adding the closing functionality to the base application. This aspect, will look for every transition having the event `getQueryConnection()` and will identify the state reached by the transition as i^{Qu}. Then, it will unify the s protocol that close connections to the data base, to the identified state.

3. *Security role management.* The application has been built in such a way that several roles have been defined for different kinds of users. For instance, there are roles for adding medical events and others for monitoring medical guarantees. Even when these functionalities are almost orthogonal, several classes of users belonging to the government need to belong to several kinds of roles. In those cases, several crosscutting security needs to be enforced, for instance:

   - Ensuring at all security-critical execution steps that the clients are obtaining a valid session and that their session are no being yielded to any other client.
   - Keeping track of the IP address of the user performing the operations and ensuring that the IP is in given in the relevant security list.

   These kinds of strategies can easily be added through aspects using our aspect language.

3.3 Semantics

We now define the semantics of our aspect language for finite-state protocols based on the original framework [6]. Due to lack of space, we cannot present the complete semantics (which includes parts of the original framework). Instead we resume the motivations of the fundamental notions of the original framework and only detail the rationale and formal definition of those parts of the semantics which are specific to finite-state protocols with identity constraints (the interested reader is referred to [10] for a complete account of the semantics). Concretely, four issues which are specific to the application of regular aspects to protocols must be addressed by a semantics for our aspect language:

- Join points. At which execution points, aspects can be mixed in within an execution of a protocol?
- Aspect weaving. When do crosscuts match join points and how actions modify component executions?
- Regular expression crosscuts. How to define this extension of the original framework?
- Protocol-modifying actions. How modifications to the structure and dynamic behavior of protocols can be defined.

Join points. The original framework is based on the formal representation of base program executions using a transition relation between join points, *i.e.*, execution points at which aspect behavior can be executed. Since aspects in the case of CwEP components modify component protocols, such join points are transitions taken during execution of the component

Weaving step

$$\frac{[j,p,p_c]^{\mathbf{sel}\ j\ A} \overset{*}{\models\!\!\!\Longrightarrow} ({p'}_s,{p'}_d) \qquad (j,({p'}_s,{p'}_d)) \rightarrow (j',({p''}_s,{p''}_d))}{(A,j,p,p_c) \Longrightarrow (\mathbf{next}\ j\ A, j', ({p''}_s,{p''}_d), p_c)}$$

Monitoring step

$$[j,(p_s,p_d),p_c]^{\emptyset} \models\!\!\!\Longrightarrow (p_s,p_d) \quad \text{[end]}$$

$$\frac{\mathcal{S}' = \mathcal{S} - \{C \rhd I\} \quad \mathcal{S} \neq \mathcal{S}' \quad C\ j = \phi \quad \neg \mathit{isAspectTransition}(j) \quad (\downarrow, \phi I, p, p_c) \overset{*}{\twoheadrightarrow} (\uparrow, \phi I, p', p_c)}{[j,p,p_c]^{\mathcal{S}} \models\!\!\!\Longrightarrow [j,p',p_c]^{\mathcal{S}'}}$$

Figure 6: Weaving regular aspects on CwEP components

protocol and the transition relation between join points defines when to proceed from one protocol transition to the next. Intuitively, the aspect transition relation must enable the definition of modifications to a protocol between two transitions. Furthermore modifications of a protocol that must be expressible involve changes to the structure of the protocol (*e.g.*, adding of new edges to the finite-state automaton representing the protocol) and changes to the dynamic state of the protocol (*e.g.*, the set of currently known component identities). Formally, if $j = (s,t,s')$ and $j' = (s',t',s'')$ are two join points, i.e., two protocol transitions, the aspect transition relation $\rightarrow$ is defined as follows:

$$((j,(p_s,p_d)),(j',({p'}_s,{p'}_d))) \in \rightarrow \quad :\Leftrightarrow \quad (j \in \delta^p \wedge j' \in \delta^{p'}) \ \wedge\ (p_s,p_d) \models t \longrightarrow ({p'}_s,{p'}_d)$$

where δ^p is the transition function of the protocol p and $\longrightarrow$ is the transition function defining the operational semantics of protocol execution. Here, protocol execution is defined in terms of modifications to the (graph) structure p_s of a protocol and its dynamic state p_d.

Aspect weaving. Intuitively, weaving can be decomposed in two steps. First, a weaving step proper which takes a set of aspects and applies it to a join point, *i.e.*, between two protocol transitions in the case of CwEP components, and thus proceeding to the next join point. Second, a (monitoring) step which defines if the crosscut of a given basic rule matches a join point j and how the corresponding action is applied to the current protocol state at j. Formally, Figure 6 defines the weaving step which determines the next join point by selecting all basic rules applicable at the current one (computed by the function **sel**) and applying them (using the function **next**)[4]. The two rules for a monitoring step define how a basic rule $C \rhd I$ is tested for applicability (through the matching operation $C\ j$) and how the action I is applied if a match occurred using the transition relation $\twoheadrightarrow$. The predicate *isAspectTransition* enables us to distinguish between transitions belonging to the original protocol and transitions introduced by aspects. This distinction is necessary because aspects can modify the structure of a protocol. Some properties, especially termination properties, can only reasonably be ensured if aspects are applied to transitions stemming from the original protocol only. This restriction is necessary, in particular, to apply interaction analyses among aspects as presented in Sec. 4.2.

[4]See [6] for the definition of the two mentioned functions.

$$\begin{array}{rcl}(C_1.C_2) \rhd I \; ; \; A & := & C_1 \rhd \texttt{skip} \; ; \; C_2 \rhd I \; ; \; A \\ (C_1 \parallel C_2) \rhd I \; ; \; A & := & (C_1 \rhd I \; ; \; A) \, \Box \, (C_2 \rhd I \; ; \; A) \\ (C_1^{+} \, C_2) \rhd I \; ; \; A & := & \mu x.((C_1 \rhd \texttt{skip} \; ; \; x) \, \Box \, (C_2 \rhd I \; ; \; A))\end{array}$$

Figure 7: Semantics of regular expression crosscuts

$$\begin{array}{lcr}(\downarrow, unify(q), (p_s, p_d))) & \twoheadrightarrow & (\uparrow, unify(q), (p_s \oplus q, p_d)) \\ (\downarrow, add(is), (p_s, p_d)) & \twoheadrightarrow & (\uparrow, add(is), (p_s, p_d[currId + I[is]]) \\ (\downarrow, send(m,i), s)) \twoheadrightarrow & (j, send(m,i), s \models (id,m) \longrightarrow s') & \twoheadrightarrow \; (\uparrow, send(m,i), s)\end{array}$$

Figure 8: Semantics of protocol-modifying actions

Regular expression crosscuts. Figure 7 defines the semantics of regular expression crosscuts (which are generated by non-terminal C in the syntax shown in Fig. 4). Intuitively, such crosscuts can be emulated in the original framework by sequences of basic rules whose actions, except that of the last rule in the sequence, do not modify the protocol state. Formally, the semantics is given by a transformation into regular aspects of the original framework (generated by non-terminal A in Fig. 4). In the semantics, `skip` represents the "empty" action. The translation is essentially straightforward but relies on one technicality. The first basic crosscuts in C_1 and C_2 must be distinct so that a translation is possible into the (deterministic) regular aspects of the original framework. (Note that this condition is part of the language definition in Fig. 4.)

Protocol-modifying actions. Figure 8 defines the semantics of the three protocol-modifying actions which are generated by non-terminal Ac in Figure 4: unify, manipulation of identity sets and emission of messages. Intuitively, a unify action adds new paths to a node of a protocol. The action of adding the identity of the component having initiated the request corresponding to the current transition modifies the dynamic state accordingly. Finally, emission of a message causes the corresponding message to be send without modification of the protocol state. Formally, the rules define the effect of the protocol-modifying actions on the protocol state (p_s, p_d) where p_s is the protocol structure and p_d the dynamic state of the protocol, *i.e.*, the set of identity sets. In the first line, application of the $unify(q)$ action results in applying q using the unify operator (see Section 2.2) to the current structural part of protocol p. In line 2, the action $add(i,k)$ adds the identity of the current component $currId$ to the current state of the identity set denoted by $is \in p_d$. Finally, the last line states that an action $send(m,i)$ denotes in a temporary transition (represented by the middle term) sending the message m to the component with identity i without modifying the protocol state (as defined by the third term).

4 Composition properties and static analysis of aspect interaction

We are now able to formally investigate composition properties involving aspects and components. In this section we consider two different kinds of property. First, three properties which are fundamental to the composition of components which are subject to modification by aspects. These properties are a finitude property of aspect-modified protocols as well as substitutability and compatibility, the two basic composition properties introduced in Sec. 2.1. Second, we show how interaction properties among aspects can be statically analyzed using an extension of the basic framework introduced in [6].

4.1 Preservation of compositional properties

Finitude. Generally, finitude of computational systems can be an issue because of unbound representations (of programs or data) or infinite executions, such as non-terminating loops. Typically, both of these problems do not arise in AOP systems because infinite representations cannot be created and infinite executions are acceptable if they do not occur as part of the weaving process but only during execution of the woven program. In AspectJ [4, 16], for instance, so-called inter-type declarararations, which can be used to introduce field definitions in classes and thus allow modification of the static structure of a Java program, cannot be applied within a loop ; woven programs may obviously be non-terminating but the AspectJ weaver is designed to be terminating on all inputs.

In the case of aspects for components with explicit protocols, two kinds of potentially harmful infinitude must be addressed.

- Since aspects may modify the structure of protocols, they may create unboiunded protocol representations.
- If aspects may apply to all parts of a protocol (including parts generated by previous aspect applications), non-terminating weaving may occur.

These two cases arise, for instance, if an aspect for control of access to a critical resource makes precede the transition performing the critical access by a transition performing a test which uses itself a security-critical operation: an unbound representation, namely an infinite path consisting of an unbound number of tests, would then be created by a non-terminating weaving operation.

There are two general methods to address these problems. The most straightforward one is to forbid actions to be applied under the scope of repetitions, e.g., under the μ-operator in our aspect language. This measure would reduce expressivity of the aspect language in a rather drastic (and probably unacceptable) manner. Another possibility is to rely on the programmer to prove that her AO programs are not subject to infinitude problems. Since this involves manual proofs, such a proceeding requires corresponding expert knowledge and is probably practical only for rather small programs.

There are, however, two variants of these methods which are appropriate in our context. First, aspects may be excluded from application to transitions introduced by other aspects. This is a reasonable solution, for example, for an aspect inserting tests for access control if the tests are safe by definition. By not being allowed to match crosscuts on such newly introduced transitions (and thus not applying actions there as well) non-terminating weaving is ruled out. Furthermore, excluding aspect application to transitions introduced by aspects also provides a solution to the problem of unbound representations.

Second, non-terminating weaving can also occur if aspects are applied multiple times to parts of the (base) protocol. This problem can be treated in many cases by exploiting properties of protocol composition operators: proofs involving finitude of aspects applied to protocols constructed using composition operators can in some (important) cases be reduced to proofs proving finitude once and for all for the composition operators. In the case of our aspect language, the *unify* operator (formally defined in [6]) can be proven to create structures which do not cause finitude issues even in case of repeated applications at the same node of a protocol, because it satisfies the following idempotency property

$$\mathit{unify}(p,\ a) = \mathit{unify}((\mathit{unify}(p.\ a)),\ a) \tag{1}$$

We are now ready to state and proof the absence of finitude problem for the aspect language shown in Fig. 4 through the following two properties:

Property 1: *Given a set S of aspects of the form defined in Fig. 4 and a protocol p of a CwEP component, weaving of S with p using the weaver defined by Fig. 6 terminates.*

Proof: Weaving does not terminate if and only if an application of the weaving steps defined in Fig. 6 is not followed by a step of the aspect transition relation $\rightarrow$. We conduct the proof by considering all cases potentially leading to non-termination as part of a double induction: an induction over transitions of the base protocol and a second on new transitions introduced by an aspect at a transition of the base protocol.

First, note that regular aspects formed using recursion ($\mu a.A$) and regular-expression crosscuts ($C_1 + C_2 \rhd Ac$) do not pose termination problems by themselves: both evaluate to infinite sequences of basic rules, each basic rule being woven at a protocol transition, that is, weaving is performed on the individual basic rules not the regular aspect as a whole.

The weaving algorithm defined by Fig. 6 allows for three potentially non-terminating computations. First, the selection function (**sel** j A) could return an infinite number of basic rules applicable at a join point. Since our aspect language (see Fig. 4) does not include any construct enabling automatic creation of aspects, only a finite number of aspects can be created. Furthermore, the number of basic rules in these aspects is also finite.

Second, crosscut matching at a join point ($C\ j$) could not terminate. However, termination of crosscut matching (essentially pattern matching of join points using disunification to allow for negation and conjunction) follows because we use the same notion of crosscuts than the original framework [6].

Third, weaving may loop because of a non-terminating application of an action, i.e., an unbounded number of transitions as part of the computation $((\downarrow, \phi I, p, p_c) \overset{*}{\twoheadrightarrow} (\uparrow, \phi I, p', p_c))$. Note that our aspect language does not allow for repetitive or recursive actions (non-terminal Ac); it is composed of finite sequences and four basic actions (*send*, *add*, *remove*, *unify*). The actions *add*, *remove*, *send* can obviously not cause non-termination because they correspond to one basic transition for the first two actions and three basic transitions for the third action (see Fig. 8). The protocol-operator *unify* is defined by a transition relying on the protocol modifying operator $\oplus$. This operator has been proven to add a finite number of states and transitions in [10].

Finally, infinite behavior through non-terminating action cannot occur because of application of aspects to aspects — *i.e.*, application of aspects to transitions introduced through *unify*-actions of other aspects — because such transitions are not subject to weaving because of the condition $\neg isAspectTransition(j)$ in Fig. 6. This completes the proof. ■

Property 2: *Given a set S of aspects of the form defined in Fig. 4 and a protocol p of a CwEP component, weaving of S with p using the weaver defined by Fig. 6 cannot result in an infinite protocol (*i.e.*, creation of an unbound number of vertices or transitions).*

Proof: The proof that one application of the weaver shown in Fig. 6 is similar to the preceding one. It essentially relies on the property, proven in [10], that a one-time application of the *unify*-action introduces only a finite number of new states and transitions.

Furthermore, the idempotency property Eq. 1 above implies that any repeated application of the unification of a given protocol with a finite-state protocol yields a finite-state protocol, that is, finitude is preserved by regular aspects. ■

Substitutability. Substitutability properties of CwEP protocols (cf. Sec. 2.1) which are modified by aspects can be derived from the substitutability properties of the actions provided by our aspect language. Protocol unification and adding of component identities yield protocols which are substitutable for the original protocols. Intuitively, both operations do not change or enlarge the set of requests permissible for any state in the protocol. Formally, the corresponding substitutability properties can be derived by an induction over all possible protocol executions directly from the substitutability properties of the two operators proven in [10].

Removal of component identities from identity sets, in general, invalidate the substitutability property of a base protocol with another protocol, because a removed identity may forbid execution of transitions in states where substitutability to the other protocol requires it to be executable.

Finally, message send actions do not change the substitutability properties of a base protocol since they do not change its state in any means.

Compatibility. Typically, compatibility between components (cf. Sec. 2.1) will not be preserved by aspects which modify protocols because such aspects are intended to modify the sequences of requests permitted by a protocol. In particular, all actions of our aspect language may invalidate compatibility. For instance, if a protocol p_1 is compatible with a protocol p_2, introduction of new paths in p_1 through unification very likely destroys this compatibility. Compatibility is only preserved in special cases, such as when the unification action introduces paths which, by chance, are compatible with already existing paths in p_2.

4.2 Analysis of interactions among protocol-modifying aspects

Besides the formalization of aspects and aspect weaving, the framework [6, 7] provides a second contribution: several static analyses of interaction properties among actions. Intuitively, these analyses allow to determine traces of base programs after which two aspects would apply actions at the same time. Formally, these analyses are defined in terms of a rewriting system performing two tasks: first, unfolding aspects (*i.e.*, finite-state automata) similar to a calculation of a product automaton; second, marking traces which end in join points triggering simultaneous application of actions of different aspects. This technique has been exploited to define different analyses depending on two characteristics. First, interactions among aspects can be analyzed with or without taking into account the base programs to which they are applied. Interaction analyses based on aspects definitions alone are frequently subject to spurious conflicts which are due to the approximations inherent in the static analyses. Analyzing conflicts among aspects only for execution traces which are valid for the base program eliminates many of such spurious conflicts. Second, interaction analysis among aspects depends on whether aspects may be applied to aspects or not.

The notion of aspects over protocols of CwEP components presented in this paper has been carefully designed to be compatible with (some of) the interaction analyses. In particular, the weaver defined in Fig. 6 is a special case of the so-called silent weaver defined in the original framework in that it does not allow aspects to be applied to parts of a protocol introduced by other aspects (by use of the predicate $\neg isAspectTransition(j)$). Furthermore, our aspect language uses terms (non-terminal T in Fig. 4) and a variable model (single-assignment) compatible with the framework. Hence, two important analyses of interactions from the framework can be applied to aspects over protocols of CwEP components: analysis of so-called strong independance of aspects from one another and weak independance of aspects *w.r.t.* a base program.

Such interaction analyses can be used in the context of the medical application SIS 2004, for instance, to determine where error handling for sessions interferes with data-base management: both can be applicable, *e.g.*, because an exception from a data-base operation would kill the current session. The interaction analyses would spot such conflicts and since the analysis yields the execution context where this interference occurs, failure recovery actions can be taken.

5 Related work

There are four types of work mainly relevant to the contributions presented in this paper: work trying to leverage synergy between aspects and components, reflective systems used for separation of dependability concerns, work on aspects and components, and approaches for reasoning about aspects. We now review these types in turn.

Up to now, very few articles have explored links between dependability properties and Aspect-Oriented Programming. Gal et al. [13], *e.g.*, use the AspectC++ system in order to modularize timing issues in a real-time system as aspects. AspectC++ (as AspectJ), however, does not support aspect definitions over sequences of execution points, nor property analysis.

A number of reflective systems have been proposed to support the separation of dependability concerns through encapsulation in meta-objects, *e.g.*, [9]. Compared to our approach, such systems typically do not support property analysis because of the use of turing-complete meta-objects and lack dedicated support at the language level.

There is a large body of work about aspects (or so-called non-functional concerns) and components, notably [8, 20, 23, 21]. All of these approaches are based on components with interfaces being composed of sets of method signatures. Hence, they do not meet the expressiveness requirements our approach aims at. Some work on components includes interfaces with (mostly finite-state) protocols, among others, work around Wright [2], SOFA [22], and component adapters [24]. However, none of these has been used as a basis of an aspect model over components.

There are very few aspect-oriented approaches which enable reasoning about properties over aspect-oriented programs. Besides the framework extended in this paper [6, 7], relevant references are [3, 14, 17]. However, none of these has been integrated with a component model, nor applied to dependability properties as demonstrated in this paper.

6 Conclusion and further work

We have motivated that potential benefits of Aspect-Oriented Programming for dependability properties of component-based software systems are hindered by insufficiently expressive aspect languages and lack of reasoning mechanisms over aspect-oriented programs. As a first step to remedy these problems, we have investigated an integration of a notion of regular aspects within a model of software components with finite-state protocols. We have formally defined an aspect language and a suitable aspect weaver enabling modification of the structure as well as the dynamic state of protocols. Furthermore, we have investigated three basic composition properties among aspects and components and have shown how two static analyses of interactions among aspects can be applied to our integrated model of components and aspects.

There are several direct leads to extend our work. First, while we demonstrated that our notion of aspects is useful to tackle certain dependability properties, the present work only constitutes a first step towards a systematic evaluation of aspect-oriented programming as support for dependability issues. Second, while the aspect language presented is useful by itself it is not complete in any sense relevant to users of the language. In particular, the set of actions in the language should be completed with additional composition operations over protocols. Third, the aspect weaver we have presented does not apply aspects on aspects: an extension to a weaver not subject to this restriction (the "visible" of [6]) should be investigated. Finally, tool support for our model is still missing (development on tool support for the underlying framework [6, 7] is currently starting).

References

[1] G. Agha, S. Frølund, R. Panwar, and D. Sturman. A linguistic framework for dynamic composition of dependability protocols. In *Proceedings of the IFIP Conference on Dependable Computing for Critical Applications*, Sicily, 1992.

[2] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–49, July 1997.

[3] J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *LNCS*, pages 187–209, 2001.

[4] AspectJ home page. `http://eclipse.org/aspectj`.

[5] A. Aviăienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January-March 2004.

[6] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, volume

2487 of *LLNCS*, pages 173–188. Springer-Verlag, October 2002. preprint version is ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-4435.pdf.

[7] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proc. of 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 141–150. ACM Press, March 2004.

[8] F. Duclos, J. Estublier, and P. Morat. Describing and using non functional aspects in component based applications. In *Proc. of the 1st international conference on Aspect-oriented software development*, pages 65 – 75. ACM Press, 2002.

[9] J.-C. Fabre and T. Perennou. Friends - a flexible architecture for implementing fault tolerant and secure distributed applications. In *Proc. of the 2nd European Dependable Computing Conference*, pages 3–20, 1996.

[10] A. Farías. *Un modèle de composants avec des protocoles explicites.* PhD thesis, École des Mines de Nantes, dec 2003.

[11] A. Farías and M. Südholt. On components with explicit protocols satisfying a notion of correctness by construction. In *International Symposium on Distributed Objects and Applications (DOA)*, volume 2519 of *LNCS*, pages 995–1006, 2002.

[12] A. Farías and M. Südholt. Integrating protocol aspects with software components to address dependability concerns. Technical Report 04/6/INFO, École des Mines de Nantes, October 2004. To appear.

[13] A. Gal, O. Spinczyk, and W. Schröder Preikschat. On aspect-orientation in distributed real-time dependable systems. The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002), January 2002.

[14] R. Jagadeesan, A. Jeffrey, , and J. Riely. A calculus of untyped aspect-oriented programs. In L. Cardelli, editor, *Proc. of the 17th Europeen Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *LNCS*, pages 54–73. Springer Verlag, 2003.

[15] G. Kiczales et al. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proc. of the 11th Europeen Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[16] G. Kiczales et al. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference, Budapest Hungary*, volume 2072 of *LNCS*, pages 327–353. Springer-Verlag, Berlin, June 2001.

[17] Shriram Krishnamurthi, Kathi Fisler, and Michael Greenberg. Verifying aspect advice modularly. In *Proceedings of ACM SIGSOFT International Symposium on the Foundations of Software Engineering (SIGSOFT'04/FSE-12).* ACM Press, November 2004.

[18] V.. Matena and M. Hapner. Enterprise javabeans^{TM} specification v1.1, December 1999. Final Release.

[19] S. Nakajima and T. Tamai. Behavioural analysis of the enterprise javabeans^{TM} component architecture. In *Proceedings of the 8th SPIN Workshop*, volume 2057 of *LNCS*, pages 163–182, Toronto, Canada, May 2001. Springer Verlag.

[20] R. Pichler, K. Ostermann, and M. Mezini. On aspectualizing component models. *Software – Practice and Experience*, 33(10):957–974, August 2003.

[21] M. Pinto, L. Fuentes, and J.M. Troya. DAOP-ADL: an architecture description languagge for dynamic component and aspect-based development. In *Proc. of the 2nd Int. Conf. on Generative Programming and Component Engineering (GPCE)*, LNCS. Springer Verlag, 2003.

[22] F. Plasil, M. Besta, and S. Visnovsky. Bounding component behavior via protocols. In *Proc. of TOOLS USA 99*, 1999.

[23] D. Suvée, W. Vanderperren, and V. Jonckers. JasCo; an aspect-oriented approach tailored for component-based software development. In ACM Press, editor, *Proc. of 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 21–29, March 2003.

[24] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Transactions of Programming Languages and Systems*, 19(2):292–333, March 1997.